UI - Umelá inteligencia

Zadanie č.1 - Zenová záhrada (1a)

Autor: Marek Čederle

AIS ID: 121193

Cvičiaci: Mgr. Matej Pecháč PhD.

Cvičenie: Štvrtok 10:00

Úvod

Ako spustiť program (primárne Windows)

Je potrebné mať nainštalovaný Python 3. Osobne som používal verziu 3.12.7, ale malo by to fungovať pre verziu 3.9 a vyššie. Je potrebné aby sme mali všetky python súbory v jednom adresáry. Otvoríme si príkazový riadok v adresári so súbormi spustíme program `main.py`. Zadáme nasledujúci príkaz:

```
> python ./main.py
```

alebo

```
> python3 ./main.py
```

Po spustení program vypíše tieto inštrukcie pre pokračovanie:

Automatický vstup:

```
Enter 0 for automatic input or 1 for manual input: 0
If you wish to print min, max, average fitness values for the generation -> enter 1, else 0: 1

Enter the minimum fitness value to be shown (from the max values): 0
```

Manuálny vstup:

```
Enter 0 for automatic input or 1 for manual input: 1
If you wish to print min, max, average fitness values for the generation -> enter 1, else 0: 1

Enter the minimum fitness value to be shown (from the max values): 0

Enter the number of rows in the matrix: 10
Enter the number of columns in the matrix: 12
Number of generations is set to infinity (it will stop when it finds solution, or when evolution count is reached)
Enter number of farmers in generation (use powers of 2 that are greater than 8, recommended 256 or 128): 256
Enter number of stones (at least 1): 6
Indexing start from 0
Possible stone positions (from assignment):
1 5
2 1
3 4
4 2
6 8
6 9
Range: (0,0) (9,11)
Enter x coordinate of stone 1: 1
Enter y coordinate of stone 1: 5
Enter x coordinate of stone 2: 2
Enter y coordinate of stone 2: 1
Enter x coordinate of stone 3: 3
Enter y coordinate of stone 3: 4
Enter x coordinate of stone 4: 4
Enter y coordinate of stone 4: 2
Enter x coordinate of stone 5: 6
Enter y coordinate of stone 5: 8
Enter x coordinate of stone 6: 6
Enter y coordinate of stone 6: 9
```

Opis problému

Máme záhradu, ktorú potrebujeme pohrabať. V záhrade sa môžu nachádzať prekážky napríklad kamene, ktoré musí farmár pri hrabaní obísť. Môže ísť však iba jedným smerom. Teda zvislo alebo vodorovne, nikdy nie šikmo. Vždy musí začať na okraji záhrady. Mimo okraju môže chodiť ako chce. Smer ktorým začne je určený jeho génmi. Ak narazí na prekážku alebo na už pohrabané miesto tak pomocou iných génov sa otočí ak má kam, ak sa nedá skúsi opačnú stranu. Ak sa nemá kde otočiť tak ukončí hrabanie. Úspešná hra je taká kde farmár pohrabe celú záhradu, prípadne maximálny počet políčok. Našou úlohou je nájsť také gény, ktoré umožnia farmárovi pohrabať celú záhradu ak je to možné. Pri nájdení riešenia vypíšeme ktorý farmár to bol, jeho gény a prechody po záhrade.

Implementácia

Na implementáciu som použil jazyk Python a nasledovné programy:

- Vývojové prostredie (IDE) – PyCharm, Visual Studio Code
- Python verzia 3.12.7
- Grafy a tabuľky – Microsoft Excel
- Dokumentácia – Markdown

Opis riešenia

Poznámka: Označenie `m_` pred premenou znamená že ide o member variable danej triedy.

Definícia záhrady:

```python
class Garden:
    m_matrix: list[list[int]]
    m_rows: int
    m_columns: int
```

- `matrix` – dvojrozmerné pole (matica pre záhradu)
- `rows` – počet riadkov
- `columns` – počet stĺpcov

Definícia génov:

```python
class Gene:
    m_genes_vector: list[list[int]]
    m_turn_genes: list[int]

    m_genes_fitness: list[list[int]]

    m_turn_genes_size: int
    m_max_size: int
    m_n: int
    m_m: int
```

- `genes_vector` – 2D pole všetkých pozičných génov jedného jedinca
- `turn_genes` – pole génov, ktoré sa použijú pri prekážkach
- `genes_fitness` – 2D pole kde sú uložené indexy génov a ich fitness hodnoty
- `turn_genes_size` – veľkosť následne vytvoreného pola turn_genes
- `max_size` – počet génov pre jedinca
- `n` – počet riadkov
- `m` – počet stĺpcov

Definícia farmára:

```python
class Farmer:
    m_genes: gene.Gene
    m_garden: garden.Garden
    m_max_genes: int
    m_farmer_number: int
    m_fitness: int
    m_stones_count: int
    m_columns: int
    m_rows: int
```

- `genes` – objekt génov pre daného jedinca
- `garden` – objekt záhrady pre daného jedinca
- `max_genes` – počet génov pre jedinca
- `farmer_number` – poradové číslo farmára v generácii
- `fitness` – hodnota fitness pre jedinca
- `stones_count` – počet kameňov v záhrade, použije sa pre veľkosť génov pre zatáčanie pri prekážkach
- `columns` – počet stĺpcov
- `rows` – počet riadkov

Definícia generácie:

```python
class Generation:
    m_max_genes: int
    m_generation_number: int
    m_farmers_count: int
    m_matrix_rows: int
    m_matrix_columns: int
    m_turn_genes_count: int

    m_fitness_through_generations: list[list[int]]
    m_farmers: list[farmer.Farmer]
```

- `max_genes` – počet génov pre jedinca
- `generation_number` – poradové číslo danej generácie
- `farmers_count` – počet farmárov v generácii
- `matrix_rows` – počet riadkov záhrady
- `matrix_columns` – počet stĺpcov záhrady
- `turn_genes_count` – počet génov pre zatáčanie
- `fitness_through_generations` – 2D pole, kde sú uložené indexy farmára a jeho fitness pre jednu generáciu

- `farmers` – pole objektov typu farmár

Program pri spustení funguje nasledovne:

Vytvorí sa prvá generácia. Na jej vytvorenie sa použijú zadané údaje či už automaticky alebo manuálne a pridá sa do pola generácií. Nasleduje vonkajší for cyklus ktorý bude iterovať do počtu zadaných generácií. Vnútorný for cyklus bude iterovať cez všetkých farmárov v danej generácii. Farmári v prvej generácii si náhodne vygenerujú gény funkciou `randomGenerateGene()`. Táto funkcia zabezpečí aby tieto gény boli unikátne a aby sa z nich dalo začať na okraji záhrady. Následne sa nastavia pozície kameňov v záhrade pre každého farmára. Potom farmár prejde (pohrabe) záhradu pomocou funkcie `walkGarden()`.

```
def walkGarden(self):
    printCounter = 1

    # initialize fitness array
    for i in range(0, self.m_max_genes):
        self.m_genes.m_genes_fitness.append([i, 0])

    for i in range(0, self.m_max_genes):
        local_x = self.m_genes.m_genes_vector[i][0]
        local_y = self.m_genes.m_genes_vector[i][1]
        hitCounter = 0
        doneWalkedLine = False
        lastX = -1
        lastY = -1

        local_dir = Dir.NONE

        # skip gene if farmer cant start walking from that position
        if self.m_garden.m_matrix[local_x][local_y] != 0:
            continue

        # get direction from which side farmer should go straight
        if local_x == 0:
            local_dir = Dir.DOWN
        elif local_x == self.m_garden.m_rows - 1:
            local_dir = Dir.UP
        elif local_y == 0:
            local_dir = Dir.RIGHT
        elif local_y == self.m_garden.m_columns - 1:
            local_dir = Dir.LEFT
        else:
            # ERROR
            return
...
```

Táto funkcia (walkGarden) funguje nasledovne:

Nastavíme si počiatočné pozície polí a ďalšie pomocné premenné. Zistíme či sa vôbec dá začať na danej pozícii. Ak nie tak ideme na ďalší gén. Ak áno pokračujeme ďalej a zistíme ktorým smerom sa má farmár vydať. Potom nasleduje while cyklus ktorý zabezpečuje aby pohrabal jednu líniu záhrady. Najskôr zistí či už nie je na konci záhrady. Ak je, tak ukončí aktuálny gén a prejde sa na ďalší. Ak nie je pokračuje v hrabaní. Ak sa farmár zacyklí tak ukončíme hrabanie a prejdeme na ďalšieho farmára. Ak sa nezacyklí pokračujeme ďalej. Nasleduje "switch" ktorý rozhoduje ako budeme zapisovať pohyb.

Ukážka "switchu" pre jeden smer:

```
...
 # switch equivalent
if local_dir is Dir.RIGHT:
    # if farmer is on the end of walkable line, terminate the while loop
    if local_y + 1 > self.m_garden.m_columns - 1:
        self.m_garden.m_matrix[local_x][local_y] = printCounter
        doneWalkedLine = True
        continue

    # if farmer can go straight
    if self.m_garden.m_matrix[local_x][local_y + 1] == 0:
        self.m_garden.m_matrix[local_x][local_y] = printCounter
        local_y += 1
    # if farmer cant go straight, change direction
    else:
        temp_dir = self.changeDirection(local_dir, self.m_genes.m_turn_genes[hitCounter %
self.m_genes.m_turn_genes_size])
        if self.canSwitchDir(temp_dir, local_x, local_y):
            local_dir = temp_dir
        else:
            temp_dir = self.oppositeDir(temp_dir)
            if self.canSwitchDir(temp_dir, local_x, local_y):
                local_dir = temp_dir
            else:
                return

        hitCounter += 1
        lastX = local_x
        lastY = local_y
...
```

Ak sa v danom smere nedá ísť ďalej tak ukončíme hrabanie línie podľa daného génu a ideme na ďalší. Ak sa dá ísť ďalej tak normálne pokračujeme inak sa treba otočiť. Pri otočení sa skúša že či sa dá ísť ďalej ak sa otočíme, ak nie skúsime opačnú stranu a ak sa nedá otočiť vôbec tak sme sa dostali do bodu z ktorého sa nedá vrátiť takže ukončíme hrabanie farmára. Toto isté sa deje pre každý smer.

V každej iterácii while cyklu si pre každý gén vypočítame hodnotu fitness. Táto funkcia má ešte jednu špeciálnu optimalizáciu a to je taká, že ak gén hrabal pri nejakom kameni, tak za každé pohrabanie pri kameni pridáme +1 k hodnote fitness. Týmto spôsobom lepšie ohodnotíme gény, ktoré mali za úlohu hrabanie na "náročnejších miestach" a budú mať vyššiu šancu na prežitie pri krížení.

```
def fitnessFunctionForGene(self, gene_print_number: int) -> int:
    done = 0

    for i in range(0, self.m_garden.m_rows):
        for j in range(0, self.m_garden.m_columns):
            if self.m_garden.m_matrix[i][j] == gene_print_number:
                # if is not on the edge
                if i != 0 and j != 0 and i != self.m_garden.m_rows - 1 and j != self.m_garden.m_columns
- 1:
                    # for each stone which is next add + 1 to indicate very good gene
                    if self.m_garden.m_matrix[i + 1][j] == -1:
                        done += 1
                    if self.m_garden.m_matrix[i - 1][j] == -1:
                        done += 1
                    if self.m_garden.m_matrix[i][j + 1] == -1:
                        done += 1
                    if self.m_garden.m_matrix[i][j - 1] == -1:
                        done += 1

                done += 1

    return done
```

Po pohrabaní (prípadnom nepohrabaní) zistíme hodnotu fitness farmára pomocou funkcie `fitnessFunction()`.

```
def fitnessFunction(self):
    done = 0

    for i in range(0, self.m_garden.m_rows):
        for j in range(0, self.m_garden.m_columns):
            if self.m_garden.m_matrix[i][j] != 0 and self.m_garden.m_matrix[i][j] != -1:
                done += 1

    self.m_fitness = done
```

Následne si zoradíme fitness génov pre budúce použitie pri krížení.

Následne pridáme index farmára v generácii a jeho fitness hodnotu do nášho pola pre fitness hodnoty v jednej generácii.

Následne testujeme že či sa nám podarilo nájsť riešenie. Ak áno vypíšeme v ktorej evolúcii, generácii a ktorý farmár je riešením a taktiež vypíšeme jeho gény a prechody po záhrade. Ak nie tak pokračujeme ďalším farmárom.

Na konci každej generácie podľa nastavenia si môžeme vypísať minimálne, maximálne a priemerné hodnoty fitness počas generácie.

Na konci vnútorného for cyklu usporiadame pole, kde sú indexy farmárov a ich fitness hodnoty.

Ak sa nám nepodarí nájsť riešenie v prvej generácii tak ideme na ďalšiu generáciu.

Pri ďalších generáciách s využitím výberu, kríženia a mutovania to bude vyzerať nasledovne.

Pre každú jednu ďalšiu generáciu sa náhodne v daný moment rozhodne ktorý spôsob výberu jedincov sa použije. Implementoval som ruletu a výber na základe miery úspešnosti.

Výber génov pomocou rulety:

```
@staticmethod
def rouletteSelection(farmer_and_fitness: list[list[int]]) -> list[int]:
    seed = time.time_ns()
    random.seed(seed)

    l_sum = 0
    fitness_of_farmer_out: list[int]
    fitness_of_farmer_out = []
    temp_in = farmer_and_fitness.copy()

    for i in range(0, len(farmer_and_fitness)):
        l_sum += farmer_and_fitness[i][1]

    for _ in range(0, len(farmer_and_fitness) // 2):
        random_num = random.randint(0, l_sum - 1)
        current_sum = 0

        for j in range(0, len(temp_in)):

            current_sum += temp_in[j][1]
            if random_num < current_sum:
                fitness_of_farmer_out.append(temp_in[j][0])
                l_sum -= temp_in[j][1]

                temp_in.pop(j)
                break

    return fitness_of_farmer_out
```

Do rulety sa ako parameter pošle pole, kde sú indexy farmárov a ich fitness hodnoty s tým že sú už usporiadané zostupne. Zistíme sumu všetkých fitness hodnôt. Následne náhodne vyberieme jedinca s tým že čím vyššia fitness tak tým má väčšiu šancu na výber. Môžeme si to prestaviť ako kruh, kde počet častí je suma fitness hodnôt všetkých jedincov a každý jedinec má šancu úmernú veľkosti jeho fitness hodnoty. Toto opakujeme kým nemáme dostatok vybraných jedincov. Vyberáme iba polovičný počet farmárov ktorý sa potom budú krížiť aby sme mali plný počet.

Výber génov pomocou hodnoty fitness na základe miery úspešnosti:

```python
@staticmethod
def fitnessOrder(fitness_and_farmer: list[list[int]]) -> list[int]:
    fitness_and_farmer_out: list[int]
    fitness_and_farmer_out = []

    for i in range(0, len(fitness_and_farmer) // 2):
        fitness_and_farmer_out.append(fitness_and_farmer[i][0])

    return fitness_and_farmer_out
```

Do tejto funkcie sa tiež ako parameter pošle pole, kde sú indexy farmárov a ich fitness hodnoty s tým že sú už usporiadané zostupne. Vyberieme iba najlepšiu polovicu.

Po výbere nasleduje kríženie jedincov:

Najskôr si vyberiem 3 rodičov z lepšej polovice predchádzajúcej generácie. Prvý rodič je najlepší, druhý je druhý najlepší a tretí je posledný z lepšej polovice. Krížim ich tak, aby som z 3 rodičov urobil 2 deti (pretože mám iba polovicu tak musím robiť 2 deti) a tým dostanem znova plný počet farmárov do ďalšej generácie. Krížim prvého rodiča s druhým rodičom a zasa prvého rodiča s tretím rodičom. Beriem vždy na striedačku jeden gén od jedného a druhý od druhého a porovnám ich ktorý je lepší a ten si vyberiem. Toto opakujem kým nedostanem plný počet génov a následne farmárov.

```python
...
# uses already sorted vector of integers
# gets indexes of half of the farmers in previous generation
# then uses those indexes to get the genes of those farmers
# then crossover
if coinFlip:
    # roulette selection
    indexes = generations[i].rouletteSelection(generations[i - 1].m_fitness_through_generations)
else:
    # fitnessOrder selection (the best by fitness)
    indexes = generations[i].fitnessOrder(generations[i - 1].m_fitness_through_generations)

# cross-over
# create new genes for farmer

for k in range(0, len(indexes)):
    # pick 3 parents, first, second, last
    parent1 = generations[i - 1].m_farmers[indexes[k]].m_genes
    parent2 = generations[i - 1].m_farmers[indexes[(k + 1) % len(indexes)]].m_genes
    parent3 = generations[i - 1].m_farmers[indexes[-k - 1]].m_genes

    for gene_index in range(0, max_genes):
        # select specific gene from parent
        pos_gene1 = parent1.m_genes_vector[gene_index]
        pos_gene2 = parent2.m_genes_vector[gene_index]
        pos_gene3 = parent3.m_genes_vector[gene_index]

        # if parent1 has higher fitness than parent2, then select gene from parent1, else from parent2
        if parent1.m_genes_fitness[gene_index] > parent2.m_genes_fitness[gene_index]:
            generations[i].m_farmers[k * 2 + 0].m_genes.m_genes_vector.append(pos_gene1)
        else:
            generations[i].m_farmers[k * 2 + 0].m_genes.m_genes_vector.append(pos_gene2)

        # if parent1 has higher fitness than parent3, then select gene from parent1, else from parent3
        if parent1.m_genes_fitness[gene_index] > parent3.m_genes_fitness[gene_index]:
            generations[i].m_farmers[k * 2 + 1].m_genes.m_genes_vector.append(pos_gene1)
        else:
            generations[i].m_farmers[k * 2 + 1].m_genes.m_genes_vector.append(pos_gene3)
...
```

Gény pre zatáčanie skopírujeme z najlepšieho farmára predchádzajúcej generácie.

Následne zavoláme funkciu na mutovanie.

```
def mutateGene(self):
    seed = time.time_ns()
    random.seed(seed)

    for i in range(0, self.m_genes.m_max_size):
        # 1/1000 probability -> 0.1%
        probability = random.randint(0, 1000 - 1)

        if probability < 1:
            geneToMutate = random.randint(0, self.m_genes.m_max_size - 1)

            # emulating do-while loop
            x = random.randint(0, self.m_genes.m_n - 1)
            y = random.randint(0, self.m_genes.m_m - 1)

            while x != 0 and y != 0 and x != self.m_genes.m_n - 1 and y != self.m_genes.m_m - 1:
                x = random.randint(0, self.m_genes.m_n - 1)
                y = random.randint(0, self.m_genes.m_m - 1)

            self.m_genes.m_genes_vector[geneToMutate][0] = x
            self.m_genes.m_genes_vector[geneToMutate][1] = y
```

Táto funkcia zabezpečí že pre každý gén má 0,1% pravdepodobnosť že zmutuje.

Ak sa riešenie nenájde ani v tejto generácii tak sa vytvorí nová rovnakým spôsobom a tak ďalej kým nenájdeme riešenie alebo ak budeme konvergovať tak sa zastavíme bez ohľadu na počet generácii a spustíme novú "evolúciu".

Evolúcia je v podstate zapúzdrenie generácií. Hierarchicky to vyzerá nasledovne:

- Evolúcia
  - Generácia
    - Farmár

Môžeme si to predstaviť, ako keby sme program spustili znovu. Gény prvej generácia sú opäť náhodné. Je to kvôli spomenutej konvergencii, aby sme nestrácali čas na hľadanie riešenia ktoré sa už nemusí dať nájsť.

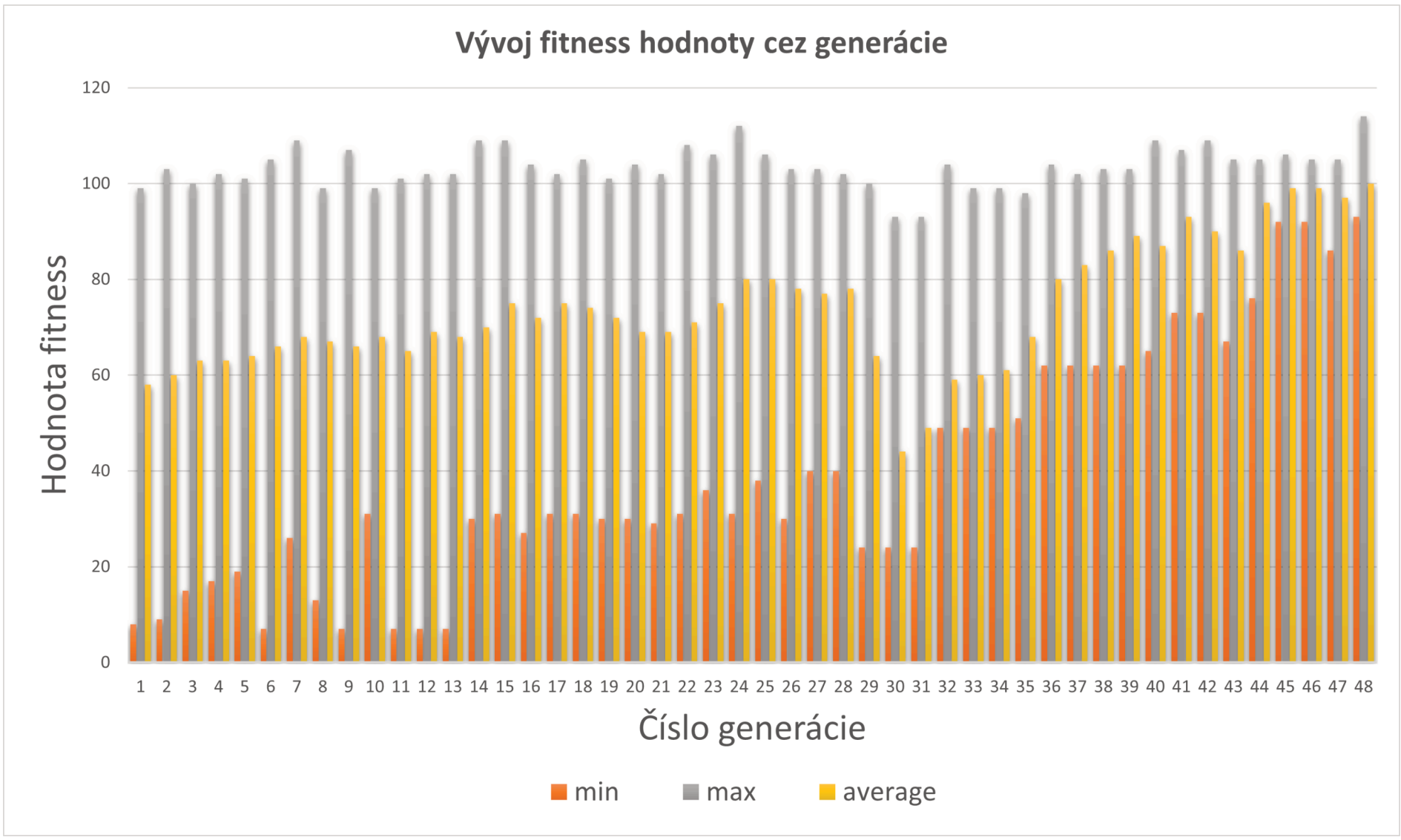
Testovanie a záver

Implementáciu som testoval primárne na nastaveniach kameňov a veľkosti záhrady, aké boli na obrázku v zadaní, ale mala by fungovať aj na iných nastaveniach. Moja implementácia vie spoľahlivo nájsť riešenie. Väčšinou to je do pár evolúcií.

Tabuľka a graf, ktoré reprezentujú vývoj fitness hodnoty jedincov cez generácie:

Generation Number	Min	Max	Average
0	8	99	58
1	9	103	60
2	15	100	63
3	17	102	63
4	19	101	64
5	7	105	66
6	26	109	68
7	13	99	67
8	7	107	66
9	31	99	68
10	7	101	65
11	7	102	69
12	7	102	68
13	30	109	70
14	31	109	75
15	27	104	72
16	31	102	75
17	31	105	74
18	30	101	72
19	30	104	69
20	29	102	69
21	31	108	71
22	36	106	75
23	31	112	80
24	38	106	80
25	30	103	78
26	40	103	77
27	40	102	78
28	24	100	64
29	24	93	44
30	24	93	49
31	49	104	59
32	49	99	60
33	49	99	61
34	51	98	68
35	62	104	80
36	62	102	83
37	62	103	86
38	62	103	89
39	65	109	87
40	73	107	93

Generation Number	Min	Max	Average
41	73	109	90
42	67	105	86
43	76	105	96
44	92	106	99
45	92	105	99
46	86	105	97
47	93	114	100



Z grafu je vidieť že na začiatku sa nám fitness postupne zväčšuje až kým nenájdeme riešenie. V tomto prípade to bolo v 48 generácii. Vidíme tam aj nejaké skoky, ktoré sú spôsobené mutáciami. Tieto mutácie však môžu byť aj príčinou prečo sa nám podarilo nájsť riešenie.

Príklad nájdeného riešenia:

```
Evolution number:  10
GenNum  Min     Max     Avg
0       8       101     60
1       12      100     61
2       14      103     63
3       22      96      66
4       3       104     66
5       16      102     65
6       18      100     64
7       26      109     67
8       21      100     63
9       24      103     66
10      24      104     69
11      32      104     70
12      20      97      62
13      30      102     64
14      31      102     68
15      29      103     68
16      26      101     68
17      26      105     71
18      29      106     74
19      28      103     76
20      41      108     79
21      48      106     77
22      44      101     78
23      32      103     76
Solution:
Evolution: 10 Generation: 24 Farmer: 80
Genes
{0, 0} {0, 7} {0, 10} {9, 2} {3, 11} {0, 8} {2, 11} {9, 8} {9, 2} {4, 11} {0, 5}
{2, 0} {0, 4} {6, 11} {7, 0} {3, 11} {0, 6} {3, 0} {6, 11} {9, 5} {6, 11} {9, 4}
Turn genes
{1} {-1} {-1} {1} {-1} {-1}
 1  9  9  9  9  9  4  2  6  6  3  5
 1  9  9  9  9  S  4  2  6  6  3  5
 1  S  9  9  9  9  4  2  6  6  3  5
 1  9  9  9  S  9  4  2  6  6  3  5
 1  9  S  9  9  9  4  2  6  6  3  8
 1  9  4  4  4  4  4  2  6  6  3  8
 1  9  4 11 11 10 10  2  S  S  3  8
 1  9  4 11 11 10 10  2  7  7  3  8
 1  9  4 11 11 10 10  2  7  7  3  8
 1  9  4 11 11 10 10  2  7  7  3  8


Found solution
```

Môžeme konštatovať že program funguje správne a nájde riešenie. Je relatívne rýchli a efektívny. Výsledky sú uspokojivé.

Jedno zo zlepšení by mohlo byť zaznamenávanie ktoré z génov pre zatáčanie boli najlepšie a použiť ich pri mutácii. Tým by sme zlepšili pravdepodobnosť nájdenia riešenia alebo nájdenie aspoň zrýchlili.

Zdroje

- Stránka predmetu UI
- Prednášky z predmetu UI (Umelá inteligencia)